

Micro Services Interview Questions

What are Monolithic Application?

A **monolithic application** is a software design where all components (UI, business logic, and database) are tightly integrated into a single, unified unit.

Key Features:

- The entire application is built as one unit.
- All parts are interdependent, making changes affect the entire application.
- Usually operates with a single, centralized database.
- The application is packaged and deployed as a single executable or archive.

Disadvantages of Monolithic Application?

- Cannot scale individual components; scaling requires scaling the entire application, which is resource-intensive.
- Components are interdependent, making it difficult to isolate and fix issues or update specific parts.
- Large codebases can slow down development, testing, and deployment.
- All components must use the same technology stack, even if better alternatives exist for specific parts.
- A small change in one module requires redeploying the entire application.
- A failure in one part can bring down the entire application.
- As the application grows, understanding and maintaining the codebase becomes harder.
- Large teams working on the same codebase can lead to conflicts and inefficiencies.

What are MicroServices?

Microservices is an approach where a large application is divided into smaller, self-contained services, each handling a specific business function. These services are loosely coupled and communicate via APIs or messaging.

Key Features:

- The app is made up of small, independent services.
- Each service manages its own data.
- Different services can use different technologies.
- Each service can be developed, deployed, and scaled separately.

Different microservices can use a different version of the same programming language.

Different microservices can use different programming languages.

Different microservices can use different architectures as well.

Why Micro Services?

In the case of monolithic applications, there are several problems such as:

1. The **same codebase** for the **presentation layer**, **business layer**, and **data access layer**. The application is deployed as a single unit.
2. **Complexity in maintenance**, and scalability becomes an issue.

Microservices solve the above problems.

- **Microservices** are a great choice when you need to **modernize** a **monolithic** or **legacy application**.
- For new software projects, microservices are ideal if the goals are to:
 - **Launch quickly** (reduce time to market),
 - Build **scalable** and **improved software**,
 - **Lower costs**,
 - Develop **faster**, or
 - Move to a **cloud-based system**.
- Each microservice is **independent**, allowing you to choose the **programming language**, **database**, and/or **architecture** best suited for it.
- Services can be **developed**, **deployed**, and **maintained separately**.

Scaling means?

scale refers to the ability of a system or application to handle an increasing amount of work or load.

- If one service in your microservice system is getting a lot of traffic (like the **order service**), you can **scale it up** by adding more instances (more servers or resources) of that service to handle the load.
- If a service doesn't require much usage, you can **scale it down** by reducing the number of instances.

What are the pros and cons of Microservice Architecture?

Pros of Microservice Architecture:

- Microservices allow each service to be developed using different technologies, frameworks, or databases based on its specific needs.
- Each microservice is designed to focus on a **single** functionality, making it easier to develop and maintain.
- Each microservice can be deployed **independently**, which allows for **frequent updates** without affecting the entire system.
- Microservices support **frequent software releases** as each service can be updated without affecting the entire system.
- Each service can have its own **security measures** (like authentication and encryption). This means if one service has a security issue, it doesn't automatically affect other services, **reducing overall risk**.
- Multiple services can be **developed and deployed in parallel**, improving overall development speed.

Cons of Microservice Architecture:

- Managing a large number of small services can be challenging and time-consuming.
- Communication between services is **complex** and can lead to increased latency.
- More effort is needed to set up, configure, and manage multiple services.
- When one service updates data, other related services might also need updates. Keeping these updates consistent across services is tricky.

→ **Example:** If you update an order's status in one service, ensuring the payment and inventory services reflect this change correctly can be complex.

- Microservices will need a large team size with right mix of experience in design, development, automation, deployment tools and testing.
- debugging of problem is harder.

when to use microservices?

- **Reduce time to market:** Teams can build and release parts of the application faster, without waiting for the whole system (complete application).
- **Scalable, better software:** You can scale only the needed services (like payment or search), making your app more efficient.
- **Lower costs:** You use resources wisely — run only what's needed, scale only where required.
- **Faster development:** Different teams can build different services at the same time.
- **Cloud-native development:** Microservices work well with modern cloud tools and platforms (like Kubernetes, Docker, etc.).
- choose microservices architecture only if the team also has experience in microservices. Do not use this architecture for simple application which can be managed by monolithic application. So you use ask yourself first do we really need microservice architecture.

What are the main features of Microservices?

- Microservices architecture breaks an application into small, independent services. Each service can be developed, tested, deployed, and scaled separately. This makes adding new features faster and easier.
- **Decentralization** - In microservices, data is decentralized. Instead of one large database (like in monolithic application), each service has its **own database** and owns its **own data**. This makes services more independent and easier to manage.
- **Black Box** - Each microservice works as a **black box**, meaning it hides its internal details from other services. Other services only interact with it through its defined API, not knowing how it works inside.

Example:

Imagine you have a **Payment Service**.





- Other services (like Order Service) just call the Payment Service API to make a payment.
- They don't need to know:
 - What payment gateway it uses
 - How it handles errors
 - How the logic works internally

They only care that:

"If I send this request, the payment will be processed."

- **Security** - In microservices, **security is handled by the platform(means API Gateway here)**, not by each individual service.

The platform takes care of things like:

-  **Certificate management** – for secure communication (HTTPS) between services —> Verifies the **identity** of a service (like a passport)
-  **Credentials** – storing and managing passwords, tokens, etc.
-  **Authentication** – checking who the user is
-  **Authorization (RBAC)** – Role-Based Access Control (e.g., only admins can access certain features)

This means **developers don't have to write custom security for every service.**

Instead, the **platform uses standard tools and rules** for security, making it **easier and safer.**

- **Polyglot** - means you can use **different programming languages, frameworks, and databases** in the same application — depending on what fits best for each service.

Different microservices can use a different version of the same programming language. Eg. Python 2.7 and Python 3.0

Different microservices can use different programming languages.

Different microservices can use different architectures as well.

How do microservices communicate with each other?

In the case of Microservice Architecture, there are 2 different types of inter-service communication between microservices.

- Synchronous communication
- Asynchronous communication

Synchronous Communication

- The **calling service waits** for a response from the other service.
- The communication happens over **HTTP** using **REST APIs**.
- The thread is **blocked** until a response is received or times out.

📌 *Real-Life Example:*

Order Service calls **Payment Service** using a REST API to process a payment.

Order Service waits for the payment result (whether payment success or not) before confirming the order.

Asynchronous Communication

- The **calling service (producer)** sends a message but **does not wait** for a response.
- It sends the message to a **message broker** (like Kafka or RabbitMQ).
- The **consumer service** listens (subscribes) to the topic or queue and processes the message later.
- The producer and consumer services are **decoupled** — they don't know or depend on each other.

📌 *Real-Life Example:*

After placing an order, **Order Service** sends a message to a Kafka topic order-created.

Notification Service listens to that topic and sends a confirmation email or SMS.

Order Service does not wait — it just sends and continues.

Message Broker Tools:

- Apache Kafka
- RabbitMQ
- Apache ActiveMQ

What is the difference between Monolithic, SOA and Microservices Architecture?

Monolithic Architecture

- Application is built as **a single unit**.
- All features (UI, logic, DB) are tightly connected.
- **Difficult to scale** parts independently.
- A change in one part requires **redeploying the whole app**.

Example:

Imagine a **restaurant** where:

- All dishes (Chinese, Indian, Italian) are cooked in **one large kitchen**.
- All chefs work together in the same space.
- If **one chef messes up**, the whole kitchen is affected.
- You **can't change or expand** one cuisine without disturbing the others.

✅ Simple to start, but becomes chaotic and hard to manage as it grows.

SOA (Service-Oriented Architecture)

- App is divided into **larger services** that **share components** (like a database or message bus).
- Services communicate via **ESB (Enterprise Service Bus)** using **SOAP/XML**.
- Focus on **reusability** across the enterprise.
- Still some **centralized dependencies** (like shared DB or ESB).

Example:

Now imagine a **hotel with a shared kitchen**, but different zones:

- There's a **Chinese section, Indian section, Italian section**, all using **the same storage, dishwashers, and waiters**.
- They are **somewhat independent**, but **still rely on shared resources** (like pantry or dishwashing area).
- If the pantry goes down, **all cuisines get affected**.

✅ Better separation, but shared parts can still cause issues.

Microservices Architecture

- App is broken into **small, independent services**.
- Each service has its **own DB**, and is **independently deployable**.
- Communication via **REST APIs** or **message brokers** (like Kafka).
- Highly **scalable, fault-tolerant**, and **technology-flexible**.

Example**:**

Now imagine **separate cloud kitchens** for each cuisine:

- A **completely different kitchen** for Chinese, Indian, and Italian.
- Each kitchen has **its own chefs, tools, storage, delivery team**, etc.
- If one kitchen fails, others **continue running smoothly**.
- They all deliver food through **a common platform (like Swiggy/Zomato)**.

✅ Fully independent, scalable, and easier to manage.

🔄 They communicate via apps (APIs/message brokers), not through a common pantry.

SOA shares and reuses as much as possible while MS focuses on sharing as little as possible

How to design Micro Services?

1. Understand the Business Domain

- Study the entire application and business process.

2. Split by Business Functionality

- Each microservice should focus on **one responsibility**.
- Examples of microservices in an e-commerce app:
 - Product Service
 - Order Service
 - Payment Service
 - Inventory Service
 - Notification Service

3. Define APIs for Communication

- Services must communicate via well-defined **APIs** (usually **REST** or **gRPC**).
- Decide between:
 - **Synchronous communication** (REST API calls)
 - **Asynchronous communication** (Kafka, RabbitMQ, etc.)

4. Decentralize Data

- Each service should have its **own database**.
- No service should directly access another service's database.

5. Design for Failure

- Use timeouts, retries, and fallback logic.
- Consider **circuit breakers** (e.g., Resilience4j).

6. Security and Authorization

- Use **API Gateway** to manage access.
- Add **JWT tokens**, **OAuth2**, and **Role-based access control (RBAC)**.

8. Observability

- Add **logging**, **monitoring**, and **tracing**.
- Use tools like:
 - **Grafana** – Shows dashboards for logs, metrics, and traces
 - **Prometheus** – Collects numbers like CPU usage, error counts
 - **Loki** – Collects logs from all services
 - **Tempo** – Captures **distributed traces** (end-to-end request flow)

9. Testing

- Unit testing each service
- Integration testing with mocks
- Contract testing between services

Ways to communicate between Microservices

We have seen Synchronous communications through -

- Rest APIs

- GraphQL
- Feign using Eureka discoveries
- GRPC (10 times faster than REST APIs) - developed by Google as substitute of REST with many more features.

Asynchronous communication usually involves some kind of messaging system like

- Active Mqs
- Rabbit MQs
- Kafka

What is Asynchronous Communication?

- In asynchronous communication, when a microservice wants to send data to another, it **sends a message** to a special middleman called a **message broker**. The broker takes care of handling the message and making sure it gets delivered safely to the receiving service.
- Once the broker gets the message, it's responsible for sending it to the receiving service. If that service is not available right now, the broker will keep trying to deliver the message until it succeeds.
- Messages can be **saved permanently** (persisted) or **kept only in memory**. If they're only in memory, they will be lost if the broker restarts before sending them to the receiver.

For an **e-commerce application**, it's best to use **persisted messages** (saved to disk) in your message broker.

- Because the broker handles delivering messages, the sending and receiving services **don't have to be running at the same time**. This means asynchronous messaging reduces the tight connection (coupling) between services that happens in synchronous communication.
- A key point is that the **sender doesn't wait** for a reply. The receiver can **respond later** with another message if needed.
- The sender just sends the message and moves on — this is called **“fire and forget.”** The receiver will handle the message when it's ready.

What if the message broker is down?

A **message broker** is a very important part of async systems, so it should be **fault tolerant** (able to handle failures).

✅ To do this, you can set up **standby replicas**. If one broker fails, another can take over (this is called **failover**).

🟡 But even with backups, failures can still happen.

So, to **make sure messages are not lost**, brokers can be set to "**at-least-once**" delivery.

This means:

- The broker sends the message to the consumer.
- The **consumer must send back an "ACK" (acknowledgment)** to say it received the message.
- If the broker **doesn't get the ACK**, it will **try again later** to deliver the message.

Types Of Async Communication?

commonly there are two types in message based communication

- point-to-point
- publish-subscribe

what is PTP Async communication?

- A **queue** is used for this type of message-based communication.
- A service that **produces** the message is called a **Producer**. It sends the message to a **queue** in a **message broker** (like RabbitMQ or ActiveMQ).
- A service that is **interested** in that message is called a **Consumer**. It listens to the queue and **consumes the message** to carry out further processing.
- Each message sent by a Producer is **consumed by only one Consumer**.
- Once the message is successfully consumed, it is **deleted from the queue**.
- If the Consumer (receiver) is **down**, the message will **remain in the queue** until the Consumer comes back up and processes it.
- This makes **message-based communication** a **reliable and resilient** approach in microservices architecture.

- **RabbitMQ** and **ActiveMQ** are popular choices for implementing this queuing system.

Publish-Subscribe (Pub-Sub) Asynchronous Communication

- - In Pub-Sub async communication, a **Topic** (or channel) is used instead of a queue.
- * A service that **sends** messages is called a **Publisher**. It publishes messages to a **Topic** in a **message broker** (like Kafka, Redis, or RabbitMQ).
- * One or more **Subscribers** (services interested in the message) subscribe to that Topic.
- * When a Publisher sends a message, **all subscribers** receive a **copy** of that message.
- * Subscribers can then process the message **independently**, and the Publisher doesn't wait for any of them — this is **asynchronous**.
- * If a Subscriber is **down**, depending on the broker and configuration, it can still receive the message later (e.g., with Kafka).
- * This pattern is useful for **broadcasting events** to multiple services (e.g., sending order event to Notification, Inventory, and Analytics services).
- * **Kafka, Redis Pub/Sub, Google Pub/Sub, and RabbitMQ (with fanout exchange)** are popular tools for implementing this.

What is Event Based Async Communication?

- A **topic** or **event stream** is used for this type of message-based communication between microservices.
- A service that **produces** an event is called a **Publisher**. It sends the event to a **topic** or **event bus** in a message broker like **Kafka, RabbitMQ (fanout/topic exchange), or Google Pub/Sub**.
- One or more **subscribing services**, called **Subscribers** or **Consumers**, listen to that event and act on it.
- Each subscriber receives a **copy** of the event and processes it **independently**.
- The Publisher does **not wait** for any response from subscribers — this is fully **asynchronous**.

- If a subscriber is **down**, the event may remain in the topic depending on the broker's **retention policy** (e.g., Kafka keeps events for a set time even after consumption).
- This pattern is ideal for **broadcasting events** to multiple services and helps build **loosely coupled and scalable microservices**.
- Common tools for this pattern include **Kafka, RabbitMQ, Amazon SNS + SQS**, and **Google Pub/Sub**.

How does synchronous and asynchronous communication work?

- Applications send messages through **function calls, service calls, or APIs**.
- The way these messages are handled (designed by the architect) impacts:
 - ⚡ Performance
 - 💻 Resource usage
 - ✅ Task execution

Synchronous Communication

- In synchronous communication, a service **waits** for a response after making a call.
- During this time, it stays **idle** until it gets the required data.


Example:

- **Vaccination Slot Booking**
 - The user tries to book a slot.
 - The system checks if a slot is available.
 - The app **waits** for the response.
 - Once confirmed, the booking proceeds.
- ✅ This prevents issues like double-booking or booking an unavailable slot.

Asynchronous Communication

- In asynchronous communication, the service **does not wait** after making a call.
- It continues with other tasks while the response or result is handled **in the background**.

Example:

- **Sending Slot Confirmation (SMS/Email)**
 - After booking, the app triggers an email/SMS.
 - It **doesn't wait** for the message to be delivered.
 - The system moves on, and notifications are handled separately.
-  This improves speed and reduces delays.

What did u use in your project and why? Pros and cons of each communication method

In my microservices project, I used **both synchronous and asynchronous communication**, depending on the use case.

Synchronous Communication

We used synchronous communication via **REST APIs** for core service interactions — for example, when the **Order Service** needed to check stock with the **Product Service** or confirm a transaction with the **Payment Service**.

Pros:

- Synchronous communication is **simple to design and implement**.
- It ensures **immediate consistency(response)** and is easy to trace during debugging.

Cons:

- It introduces **tight coupling** between services.
- One major drawback is the **risk of cascading failures** — if one service goes down, it can affect the whole request chain.

 **To mitigate this**, we implemented:

- **Service discovery** and **load balancing** using Spring Cloud and Eureka.
- **Circuit breakers** and **retries** using Resilience4j to improve fault tolerance.

Asynchronous Communication

For background tasks, we used asynchronous communication via **Kafka** — for example, sending notifications after an order was placed, or for analytics and inventory updates.

✅ Pros:

- Asynchronous communication offers **better scalability** and **resilience**.
- It **decouples services**, so they can operate independently and handle failures gracefully.

❌ Cons:

- It's **more complex to implement**, especially in terms of **message tracking** and **ensuring consistency**.
- There's also **eventual consistency**, meaning updates may not reflect immediately.

Note:

Circuit Breaker = When Service A calls Service B and B is slow or failing, you wrap the call with a **Circuit Breaker** to prevent overload.

Resilience = The ability of a system to **handle failures gracefully** and **recover quickly** without affecting the overall functionality.

eventual consistency

1. **Order Service** places an order and sends an **event** (OrderPlaced) to Kafka.
2. **Inventory Service** listens to this event and **reduces stock**.
3. There may be a **small delay** between placing the order and the inventory update.

👉 During that delay, the data across services is **inconsistent**.

But eventually — once the event is processed — **all services reflect the same data**.

When to use which communication method?

When you're starting to build a new application, it's better to use **synchronous communication**. It's easier to understand, faster to build, and helps you develop features quickly. For example, when Service A calls Service B and waits for a response — this makes development and testing simple in the beginning.

But as your application grows and becomes more complex, with many microservices interacting, synchronous communication can become a problem. If one service is down or slow, it can affect all the other services that depend on it.

At that stage, you should look at **asynchronous communication**. That means services communicate by sending messages through a message broker (like Kafka, RabbitMQ, or ActiveMQ). The sender doesn't wait for a response, and the receiver processes the message whenever it's ready. This makes the system more scalable, fault-tolerant, and independent.

To decide when to use async:

- Look at how your services interact.
- If a service **doesn't need an immediate response** to move forward, you can make that communication asynchronous.

Example:

- Placing an order = synchronous
- Sending an order confirmation email = asynchronous

So, start with sync for simplicity, and move to async when needed for better performance and stability.

Why SAGA Design Pattern? What Problem Does It Solve?

The Saga pattern helps to solve data consistency problem in microservices when one task is handled by many services and something goes wrong during the process.

- In **monolithic applications**, we had **one database** and could use **ACID transactions** to **rollback** everything if one step failed.
- But in **microservices**, each service has its **own database**.

So, **we can't use a single database transaction** to rollback across multiple services.

Example – Food Delivery App (Swiggy/Zomato):

In Monolithic:




- One DB → Orders, Payments, Deliveries tables.

- All actions happen in **one transaction**.
- If payment fails, order creation is also rolled back.

In Microservices:

- **Order Service** → accepts the order
- **Payment Service** → processes the payment
- **Delivery Service** → manages delivery

Now imagine:

- Payment is successful 
- But **no delivery partner** is available 
- Now you paid, but you won't get the food 

This creates **inconsistency**, and we need to **undo the payment** and **cancel the order**.

- Saga handles this situation by creating a **series of smaller transactions** in each service.
- If a failure occurs in one service (like Delivery), it **calls compensating actions** in the previous services (like refund in Payment, cancel in Order).
 - So, even **without a global transaction**, the system can **stay consistent**.

Global Transaction:

A **global transaction** means a **single transaction** that includes **multiple operations across multiple services or databases**, and either **all of them succeed**, or **all are rolled back** if one fails — just like in a monolithic application.

In monolithic apps, this is easy because all the operations happen in one place and one database, so rollback is simple.




But in **microservices**, each service has its **own database**, so you can't use a single global transaction to manage everything. That's why we need patterns like **Saga** to keep things consistent **without** relying on global transactions.

How SAGA DP handles failures of any individual saga

When a **Saga** fails at any step (in one of the services), it uses something called a **compensating transaction** to **undo** the changes made by the previous successful steps.

Simple explanation:

Imagine you are placing an order:

1. **Order Service** → creates the order 
2. **Payment Service** → payment done 
3. **Delivery Service** → fails  (No delivery partner)

At this point, the **Saga detects failure** in the Delivery step.

Now what happens?

- It tells **Payment Service** to **refund** the money (compensating action)
- It tells **Order Service** to **cancel** the order (another compensating action)

These compensating transactions are **defined ahead of time** for each service.

So, Saga handles failure by **reversing** previous steps using those predefined compensating actions.

This way, the system stays **consistent**, even when part of the process fails.

Ways to implement SAGA?

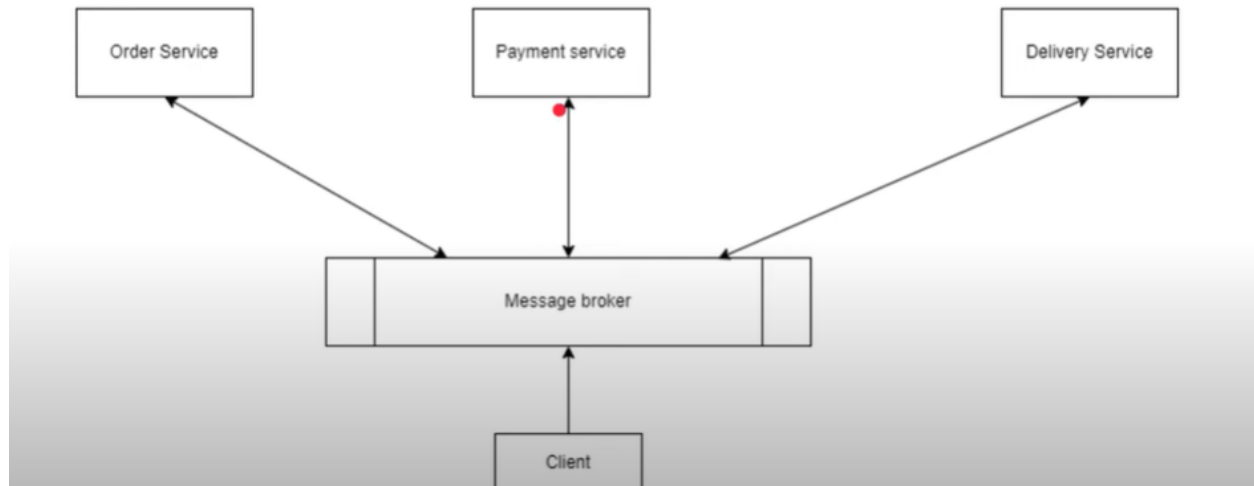
There are two type of saga implementation ways

- choreography
- orchestration

What is Choreography Saga Pattern?

- It's a **decentralized way** to manage a long-running business process across multiple microservices.
- **No central coordinator.**
- Each microservice performs its **local transaction** and then **publishes an event** (like OrderCreated).
- Other services **listen** to that event and do their own work in response.
- All communication is done via a **message broker** (e.g., Kafka, RabbitMQ).

Choreography Saga Pattern



Advantages:

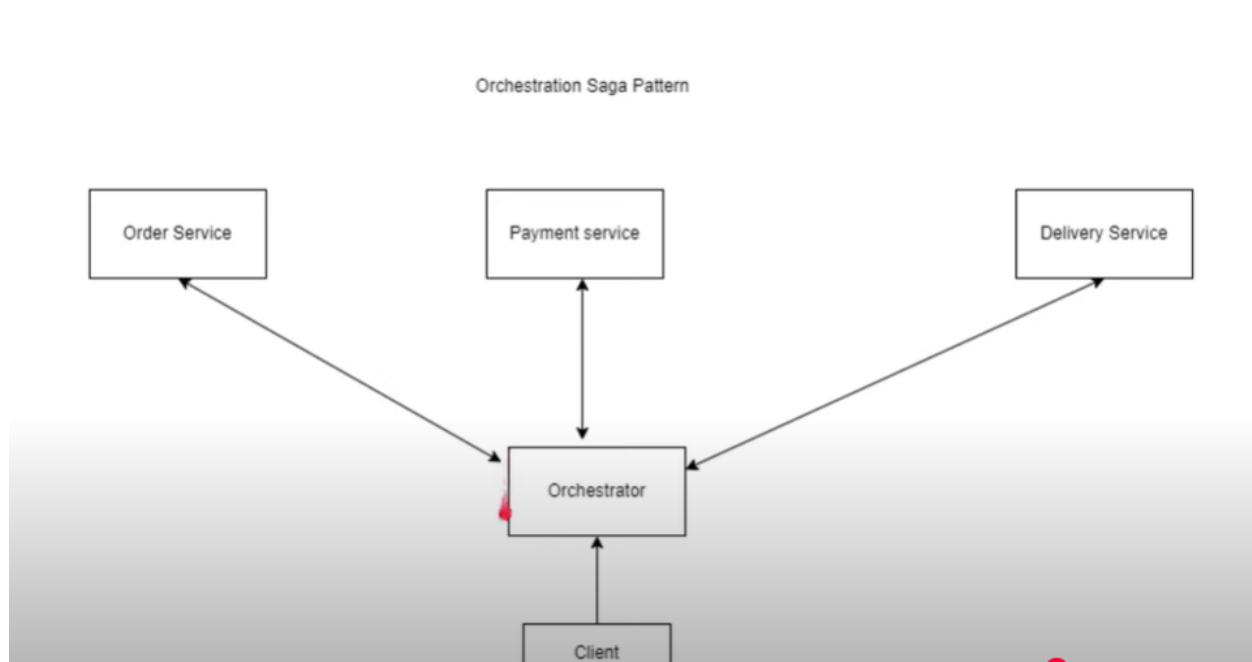
- **Simple to implement** for small workflows.
- **No extra service** (like an orchestrator) needed.
- **No single point of failure** — services are loosely coupled and independent.
- **Easier to scale**, since each service handles its part.

Disadvantages:

- Becomes **hard to manage** as the workflow grows — it's tough to track who listens to what.
- Risk of **cyclic dependencies**, since services may rely on each other's events.
- **Debugging and monitoring** the flow is more difficult, especially with many services involved.

What is Orchestration Saga Pattern?

- It's a **centralized approach** to manage workflows across microservices.
- A **saga orchestrator** (a separate service) coordinates the entire process.
- The orchestrator tells each microservice **what to do** (e.g., "create order", "process payment").
- Services **execute their local transaction** and respond with a success or failure.
- The orchestrator **controls the flow** based on these responses.



Advantages:

- Great for **complex workflows** with many steps or services.
- **Easy to manage and track** — clear control over the process.
- **No cyclic dependencies** — services don't talk to each other directly.
- Services are **simpler**, as they just do their task and respond.

Disadvantages:

- Adds **design complexity** — you need to build and maintain the orchestrator.
- Introduces a **single point of failure** — if the orchestrator goes down, the whole workflow can break.

How to design and implement a microservice?

1. Start with a Monolith

- Begin with a **simple monolithic app** to quickly build and test core features.
- It helps understand the full business logic before breaking things apart.

2. Organize Teams Properly

- Align **one small, independent team per microservice**.

- Each team should **own** the full lifecycle: design, build, deploy, and maintain.

3. Split the Monolith Gradually

- Identify **logical boundaries** in the monolith (e.g., Order, Payment, Inventory).
- Slowly extract those into **separate services** with their own codebase and database.

4. Design for Failure

- Assume services **can fail** and design accordingly.
- Use **circuit breakers, retries, timeouts, and fallbacks** to handle failures.

5. Add Monitoring and Logging

- Use tools like **Prometheus, Grafana, Loki, Tempo** for observability.
- Helps in tracking errors, performance, and inter-service communication.

6. Implement CI/CD

- Use **Continuous Integration & Continuous Deployment** to automate testing and deployment.
- Speeds up development and makes releases **faster and safer**.

Can you please provide examples of situations where using microservices over monolithic systems and vice versa is a better choice?

Scenarios where monolithic architecture is preferred over microservice Architecture

App is Small or Simple → It's easier to build, test, and deploy as a single unit.

Example: A basic blog, a school management system, or a personal finance tracker.

Faster Development is Needed → Everything is in one place, so you can build features faster.

Small Team → Easier to coordinate and avoid the overhead of managing multiple services.

Limited Resources (Infrastructure or People) → Monoliths require less setup, hosting, and monitoring.

Better Performance Needed → Function calls in a single codebase are faster than inter-service communication.

Example: High-speed in-memory processing app

Simple Business Logic → No need to complicate it with distributed systems.

Scenarios where microservice architecture is preferred over monolithic Architecture

App is Large and Growing → You can scale individual services like Orders, Payments, or Inventory separately.

Example: E-commerce platforms like Amazon or Flipkart.

Large Teams Working in Parallel → Each team can own and work independently on different microservices.

Flexibility in Tech Stack Needed → Microservices allow you to choose the best tool for each job.

Example: One service in Java, another in Node.js.

Frequent Updates & Fast Deployments → You can deploy services independently without affecting the whole system.

Example: Apps needing continuous delivery and rapid feature releases.

Complex Business Logic / Domains → Helps manage complexity by breaking down the app into domain-driven services.

High Availability and Resilience Needed → Failures in one service don't crash the entire system.

Final Analogy:

- **Monolith = Big cake** 🍰 – Easier to bake, serve, and eat when small.
- **Microservices = Cupcakes** 🧁 – Better when serving a crowd with different tastes and needs.

How do you make sure a Microservices-based application can handle more users as it becomes more popular?

1. Load Balancing

👉 **Problem:** Too many users are using your app → one server gets overloaded.

👉 **Solution:** Add a **Load Balancer** in front of multiple instances (copies) of your service.

🧠 **How it works:**

If 1,000 users hit your app, the Load Balancer splits them between 3 servers:

- 1st server gets 333
- 2nd server gets 333
- 3rd server gets 334

📌 **Tools:**

- **NGINX, Kubernetes Service, Spring Cloud LoadBalancer**

2. Horizontal Scaling

👉 **Problem:** Your Order Service is slow when too many users try to place orders.

👉 **Solution:** Add more instances of Order Service.

🧠 **How it works:**

Initially, you had 1 instance (container/pod). You scale to 5 instances → Now 5 pods can handle orders in parallel.

📌 **Tools:**

- **Kubernetes** (`kubectl scale deployment order-service --replicas=5`)

3. Auto Scaling

👉 **Problem:** You don't want to manually scale up every time traffic increases.

👉 **Solution: Automatically add/remove** instances based on CPU or memory usage.

🧠 **How it works:**

If CPU usage goes above 70%, the system automatically adds more instances.

When traffic drops, it removes them to save cost.

📌 **Tools:**

- **Kubernetes HPA, AWS Auto Scaling**

4. Caching

👉 **Problem:** Users keep asking for the same data (like product list) → database becomes slow.

👉 **Solution:** Store frequently-used data **in memory** (cache) instead of hitting DB every time.

🧠 **How it works:**

1st request → fetch from DB → store in Redis

Next requests → return data from Redis → very fast!

📌 **Tools:**

- **Redis, Memcached, Spring Boot Cache**

5. Database Scaling

👉 **Problem:** Your database becomes a bottleneck (slow or overloaded).

Solutions:

- **Vertical Scaling:** Add more CPU/RAM to DB
- **Read Replicas:** Main DB handles writes, replicas handle reads
- **Sharding:** Split one large DB into smaller DBs (per region, per user, etc.)


🧠 **Example:**

User data is read-heavy → use replicas to offload read operations.

Tools:

- PostgreSQL Replication, MySQL Read Replicas, MongoDB Sharding

6. Asynchronous Processing

 **Problem:** Some tasks take time (e.g., sending confirmation email, updating stock) and slow down the response to user.

 **Solution:** Do them in background using a **message queue**.

How it works:

- User places order → order saved quickly
- Background workers handle email, stock update using Kafka/RabbitMQ

Tools:

- Kafka, RabbitMQ, Spring Boot + @Async

How do you handle data consistency in microservices?

1. Synchronous Communication

- One service calls another using REST or gRPC.
- Ensures strong consistency, but tightly couples services and can lead to failures if one is down.

Example: Order Service directly calls Payment Service and waits for response.

2. Asynchronous Communication (Event-Driven)

- Services communicate via events using a message broker like Kafka or RabbitMQ.
- Promotes loose coupling and better scalability.
- Achieves **eventual consistency**.

Example: OrderCreated event triggers Payment Service → emits PaymentSuccess → triggers Inventory Service.

3. Saga Pattern (Best practice for distributed transactions)

- Splits a big transaction into smaller local ones.
- Each service performs its own operation and publishes an event.
- Two types:
 - **Choreography** – each service listens and reacts to events (no central controller)
 - **Orchestration** – a central saga coordinator manages the flow.

Example:

Order → Payment → Inventory → Notification

If Payment fails, emit PaymentFailed and rollback the Order.

4. CQRS (Command Query Responsibility Segregation)

- Separate models for **write** (commands) and **read** (queries).
- Helps scale and optimize performance, especially for read-heavy apps.

5. Event Sourcing

- Instead of saving the final state, store all state changes as events.
- System state is built by replaying events.

Helps with audit logs, history, and rollback.

6. Monitoring & Logging

- Use tools like **Grafana, Prometheus, Tempo, Loki, ELK** to monitor flows and detect failures.
- Helps quickly identify consistency issues across services.

7. Conflict Resolution & Idempotency

- Ensure that operations can handle retries safely (idempotency).
- Use techniques like **last-write-wins**, **versioning**, or **manual resolution** to resolve data conflicts.

Note: **Idempotency** means:

"No matter how many times you perform an operation, the result stays the same."

If you **repeat the same request**, it should not have any additional effect.

What is difference between SAGA and 2 Phase Commit? Which one will you prefer and why?

Payment in E-Commerce

2PC:

- Order → Payment → Inventory → Commit
- Coordinator waits for all to say "ready" (Phase 1)
- If all are ready, it sends "commit" (Phase 2)
- If one fails → everything rolls back

Problems:

- Services blocked until all vote
- Not fault-tolerant
- Bad for distributed systems

Saga:

- Order Service → local transaction
- Emits event → Payment Service
- Payment → emits success/failure
- Inventory Service → reserve or rollback
- If Payment fails → Compensation: cancel order

Each step commits immediately, no locking!

Saga	2 Phase Commit (2PC)
Eventual consistency	Strong consistency

Uses compensating transactions	Uses single rollback
Decentralized or orchestrator-based	Centralized coordinator
Scalable and non-blocking	Not scalable, blocking
Fault-tolerant	Not fault-tolerant
Loosely coupled services	Tightly coupled services
Recommended in microservices	Not recommended in microservices

Banking (strict money transfers) -> 2PC (rarely used) -> Strong consistency needed, but often replaced by Saga + Outbox Pattern.

Which One to Prefer?

"I prefer the **Saga pattern** for microservices because it's designed for distributed systems. It avoids locking, is more fault-tolerant, and works well with asynchronous communication.

Two Phase Commit is tightly coupled and not scalable, so it's not suitable for modern cloud-based microservices."